Modern JavaScript

Arrow Functions

A compact alternative to a traditional **function expression**, but **is limited** and can't be used in all situations.

- Does not have its own bindings to **this** or **super**, and should not be used as **methods**.
- Does not have **arguments**
- Not suitable for **call**, **apply** and **bind** methods, which generally rely on establishing a **scope**.
- Can not be used as **constructors**.

Arrow Functions

Arrow functions allow us to write shorter function syntax

```
// Traditional Function Expression
const f1 = function (a){
 return a + 100;
}

// Arrow Function Break Down

// 1. Remove the word "function" and place arrow
between the argument and opening body bracket
const f2 = (a) => {
 return a + 100;
}

// 2. Remove the body brackets and word "return" --
the return is implied.
const f3 = (a) => a + 100;

// 3. Remove the argument parentheses
const f4 = a => a + 100;
```

Template literals

Template literals are string literals allowing:

- embedded expressions.
- multi-line strings.
- interpolation features.

```
let person = "Mike";
let age = 28;

let text1 = `Hello ${person},
how are you doing this fine morning?`;
console.log(text1);

// Tagged templates
function myTag(strings, personExp, ageExp) {
 let str0 = strings[0]; // "That "
 let str1 = strings[1]; // " is a "

 let ageStr = ageExp > 99 ? "centenarian" : "youngster";

 return `${str0}${personExp}${str1}${ageStr}`;
}

let text2 = myTag`That ${person} is a ${age}`;
console.log(text2);
```

Conditional (ternary) operator

(condition ? ifTrue : ifFalse)

The conditional operator returns one of two values based on the logical value of the condition.

```
let person = {
 name: "tony",
 age: 20,
 driver: null,
};

// if statement
if (person.age >= 18) {
 person.driver = "Yes";
} else {
 person.driver = "No";
}

// Now, the ternary operator:
person.driver = person.age >= 18 ? "Yes" : "No";
```

Binary logical operators

- && Logical AND.
- || Logical OR.
- ?? Nullish Coalescing Operator.

```
false || console.log("printed");
true || console.log("not printed");

true && console.log("printed");
false && console.log("not printed");

//The nullish coalescing operator (??) is a
logical operator that returns its right-hand side
operand when its left-hand side operand is null or
undefined, and otherwise returns its left-hand
side operand.
null ?? console.log("printed");
0 ?? console.log("not printed");
```

Optional Chaining (?.)

Optional chaining syntax allows you to access deeply **nested object properties** without worrying if the property exists or not. If it exists, great! If not, **undefined** will be returned.

```
let person = {
 firstName: "Daniel",
 lastName: "Smith",
 address: {
   city: "Oradea",
 },
 fullName: function () {
   return `${this.firstName} ${this.lastName}`;
 },
 cars: [],
};

// obj?.prop
const city = person?.address?.city;

// obj?.[expr]
const firstName = person?.["firstName"];

// arr?.[index]
const car = person.cars?.[0];

// func?.(args)
const fullName = person.fullName?.();

// Combining with the nullish coalescing operator
const street = person?.street ?? "Unknown street";
```

Default function parameters

Default function parameters allow named parameters to be initialized with default values if **no value** or **undefined** is passed.

```
function multiply(a, b = 1) {
 return a * b;
}

console.log(multiply(5, 2));
// expected output: 10

console.log(multiply(5));
// expected output: 5
```

Array/Object Destructuring

The destructuring assignment syntax is a JavaScript expression that makes it possible to **unpack values** from arrays, or properties from objects, into **distinct variables**.

```
function getNumbers() {
 return [1, 2, 3];
}

let tmp = foo();
let val = tmp[0];
let b = tmp[1];
let c = tmp[2];

// Destructuring array
let [a, b, c] = foo();

// Destructuring object
let { x: someX, y: someY, z: someZ } = {x: 4, y: 5, z: 6,};

// You can even solve the traditional "swap two variables"
task without a temporary variable:
let x = 10;
let y = 20;
[y, x] = [x, y];
```

Rest/Spread Operators ...

... operator that's typically referred to as the spread or rest operator, depending on where/how it's used.

```
// Array Rest
const [a, b, ...restNumbers] = [1, 2, 3, 4, 5];
console.log({ a, b, restNumbers });

// Object Rest
const { name, ...restInfo } = {
 firstName: "Daniel",
 lastName: "Smith",
 age: 40,
};
console.log({ name, restInfo });

// Spread Array
let numbers = [3, 4, 5];
let allNumbers = [1, 2, ...numbers];
console.log(allNumbers);

// Spread Object
let info = {
 lastName: "Smith",
 age: 40,
 city: "Oradea",
};

let person = { firstName: "Daniel", ...info };
console.log(person);
```

Resources

- https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Spread_syntax
- https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Destructuring_assignment
- https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Template_literals
- https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/Arrow_functions